Transparent Distributed Programming under Linux

Kamran Karimi
School of Computer Science
University of Windsor
Windsor, Ontario
Canada N9B 3P4
kamran@uwindsor.ca

Mohsen Sharifi
Department of Computer Engineering
Iran University of Science and Technology
Narmak, Tehran
Iran 16846-13114
msharifi@iust.ac.ir

Abstract

Developing parallel and distributed programs is usually considered a hard task. One has to have a good understanding of the problem domain, as well as the target hardware, and map the problem to the available hardware resources. The resulting program is often hard to port to another system. The development and maintenance process may thus be costly and time-consuming. In this paper we propose giving priority to hiding the details of distributed programming behind normal data sharing and synchronisation mechanisms. The programs are thus written as if they are meant to be run on a single, parallel computer. The developers still have to divide the task and make sure that the problem is solved in parallel, but the details of data transfer and synchronisation over a network are hidden. The resulting programs can thus be developed on non-distributed, non-parallel environments, and then be run on a variety of distributed and/or parallel platforms. Though such programs may not be as optimised as programs written specifically for a distributed computer, the speedups in programming time and costs may offset the losses. In this paper we introduce Distributed Inter-Process Communication (DIPC), a heterogeneous distributed programming system that hides inside Linux's kernel, traps access requests to System V IPC mechanisms (messages, semaphores and shared memories) and delegates them for execution on other computers as needed. The results are then handed back through the kernel to the calling process, which is unaware of any distributed activity. DIPC currently supports Intel, PowerPC, ALPHA, MIPS, SPARC and Motorola 68K processors.

1. Introduction

Programming a distributed system often needs a good understanding of the low-level features of the hardware system. Code then must be written specifically for the target distributed system. The good thing about this scheme is that it is possible to gain the maximum performance. The negative aspect is that programming is hard and needs experts that know the problem domain as well as the target hardware. The resulting programs may not be portable to another distributed system without going through a major phase of re-design and debugging.

Transparent distributed computing hides the distributed aspects of the system from the applications. In this paper we introduce Distributed Inter-process Communication (DIPC) [3, 4]. DIPC is a software system to create a multi-computer from Linux [10] machines connected together over a TCP or UDP [7] network. It eases data exchange and synchronisation between programs, and uses the CPU's Memory Management Unit (MMU) [5] to provide Distributed Shared Memory (DSM) capabilities [9]. DIPC provides parallelism at the program level. The user is responsible for invoking remote programs in computers with appropriate machine architectures. There is basic support for heterogeneous environments.

Currently DIPC has been implemented on Linux 2.2.x kernels, and can run on Intel i386, Motorola 680x0, ALPHA, PowerPC, SPARC and MIPS processors. Information on obtaining DIPC and ideas for further work on it appear at the end of this paper.

DIPC works by making UNIX System V's Inter-Process Communication (IPC) [8] mechanisms distributed. It offers distributed semaphores, distributed messages and distributed shared memory segments. Distributed semaphores allow remote processes in DIPC to synchronise their access to shared resources such as a distributed shared memory. Distributed messages allow processes to exchange information in the form of byes of data. A DSM can be used to asynchronously transfer data between processes.

DIPC traps System V IPC mechanisms in the local kernel. When a process makes a request to use a semaphore or a message queue, DIPC examines the call and if necessary, invokes an operation on another machine. The request is thus executed remotely and

the results are sent back to the originating machine's kernel. DIPC then hands the results back to the local process, which has been sleeping inside the kernel, unaware of any distributed activity. The shared memory mechanism does not require the local process to invoke a system call (it is an asynchronous mechanism), so DIPC uses the MMU to restrict access to individual pages, or a whole segment, of the shared memory. Any invalid access is then trapped inside the kernel, and DIPC determines whether the process requires remote access to the shared memory. If so, the contents of the shared memory are moved over the network to satisfy the local process' request. When the contents of the shared memory are obtained, the local process is allowed to continue. This mechanism is much like that of swapping memory out to a secondary storage device, except that DIPC may move the memory contents to another machine.

DIPC has already been integrated into other systems that provide distributed services. For example, DIPC has been added to a Beowulf [2] cluster that runs the PANTS Application Node Transparency System (PANTS) suite of programs. PANTS is designed for a Beowulf cluster of Linux computers to provide automatic and transparent load sharing of processes [1].

The rest of this paper is organised as follows. Section 2 briefly describes how a DIPC application is written and run. Section 3 introduces more details about DIPC's workings. In Section 4 we introduce the notion of a DIPC cluster, which is made of a number of computers that collaborate together. Section 5 introduces the user-space component of DIPC, called *dipcd* (the DIPC daemon). In Section 6 we discuss DIPC's transparent data transfer. DIPC's implementation of distributed shared memory is presented in Section 7. Section 8 produces the results of a number of experiments meant to measure the data-transfer frequency in DIPC. Section 9 concludes the paper and proposes a number of areas for future work and collaboration.

2. A DIPC Application

A distributed program that uses DIPC has a model similar to a parallel application, except that no threads are to be used (unless support for distributed threads is provided by another package). Two or more processes are started in the same computer or remotely, and one of them creates the needed IPC resources using well-known integer values known as *keys*. The machine on which an IPC structure is created is called the *owner*. Other processes gain access to these resources using the same key values. Processes can now exchange data and synchronise by calling the IPC system calls or read and write the shared memory.

A DIPC application signals that the created IPC structures will be used over a network using a flag, IPC_DIPC, when creating the IPC structures. This flag is used to retain backward compatibility with the existing Linux programs that use System V IPC. As a result, the default behaviour of a Linux kernel which is enhanced with DIPC functionality is to treat the IPC structures as local.

When a process calls an IPC system call, the kernel parts of DIPC determine if the current machine is the owner of the IPC structure and if not, inform the user-space components of DIPC (in *dipcd*), which then gather the needed parameters and send a request to the owner machine. The request is received by the user-space parts of DIPC in the owner machine and executed remotely in the owner kernel. Any results are then sent back to the original computer, passed back to the kernel and then to the original process that had made the system call. More information on this sequence of operations comes later in the paper.

After setting up a shared memory, there are no synchronous system calls for shared memory reads and writes. A shared memory is first considered to be read-only, and its contents are duplicated on any computer that needs read access. When a process wants to write to a shared memory, the access is trapped and the contents of the shared memory on other machines, if any, are set to be invalid. The writing process is then allowed to proceed with the operation. When another read or write request arrives in another machine, the contents are copied to that computer. More details on handling DSM come later.

The user process is not made aware of any of these activities because it still requests a service and receives the results through the local kernel. For seeing example source code that uses the System V IPC for data exchange in a DIPC cluster, and details on how a DIPC application is written, refer to [4].

3. DIPC Work Process

DIPC has two parts: the main part, *dipcd*, runs in user space as an ordinary process with root privileges and does the decision makings and remote data transfers. Such a design simplifies the kernel parts of DIPC, making a port to other kernel versions or processors easier. The other part of DIPC is inside the kernel, and provides the first part with necessary functionality and information to do its job. It is not possible to run *dipcd* on a Linux kernel with no DIPC support.

The kernel parts of the DIPC are short-circuited when the *dipcd* program is not running. In the

absence of *dipcd*, DIPC system calls in a program behave as if they are normal (local) System V IPC calls. *dipcd* itself uses ordinary means to access System V IPC mechanisms. These mechanisms are changed in the kernel, so they treat *dipcd* differently than other user processes. For example, *dipcd* can gain access to a shared memory segment by a *shmget()*, even when it has been removed (*shmctl()* with the IPC_RMID command), but not deleted from the kernel. All manipulations by *dipcd* to IPC structures are done locally, without them being visible outside that machine. This difference is in contrast to normal user processes, where actions on distributed IPC structures may affect other computers in the network.

DIPC is only concerned with the transfer of data in a distributed environment. Starting suitable programs in different computers is left to the user. This means that the programs to be executed may need to be present in the computer on which they are to execute. Programs could be placed in different computers once, and used many times after that. This implies that there is no overhead for transferring code in the network whenever a program is started.

There are two kinds of activity in (D)IPC: First, synchronous: here the programmer uses system calls to carry out an action. Examples include using a system call such as *msgget()* to gain access to a message queue IPC structure, or using the *msgrcv()* system call to receive a message. The program makes the call and waits for its completion before continuing. DIPC will always take some action here. The second kind is asynchronous activity: Reading and writing a shared memory may cause an asynchronous action to take place. The programmer can not predict if and when such an activity may take place. One example is reading from a shared memory, when the relevant pages are swapped out (IPC) or are not in the requesting machine (DIPC). The needed pages are fetched from wherever they are and the program resumes execution. These events may or may not occur for each reference to the shared memory.

4. DIPC Clusters

A DIPC distributed system consists of a number of computers connected over a TCP/IP or UDP/IP network. Some or all of these machines could be in one *cluster*. There may be more than one cluster in a physical network, but each machine can belong to at most one of them. Clusters are logical entities: they can be created or removed, or their members changed without the need to change any of the network's physical properties.

Computers on the same cluster can use DIPC to transfer data and synchronise themselves without interfering with the workings of machines on other clusters at all, even though they may also be using DIPC and even the same applications. In other words, as far as DIPC is concerned, computers never interact in any way with machines outside their own cluster.

It is the ability to exchange data between programs running on different machines in a cluster that makes DIPC a distributed system, but it is also possible to run all processes of a DIPC-enabled application in a single machine. They behave as if they are using normal System V IPC. This is because there is no explicit reference to any particular computer in DIPC. The same program may use different computers during different invocations to complete its job, freeing the program of being dependent on certain machines with certain addresses. This also means that programmers can use single machines to develop their application, and later run it in a multi-computer cluster. In other words, the user can do the final mapping between the resources needed by a program, and the physical resources available.

Processes in a cluster can use the same key to access an IPC structure. This structure should first be created. This is done by one of the *xxxget* system calls (*shmget()*, *semget()* or *msgget()*) [8]. After creation and initialisation, other processes, possibly in other machines, may be able to access this structure. So a key here can have the same meaning in all the computers of the cluster. Put another way, computers in the same cluster have a common DIPC key space.

Many legacy software with no accessible source code may use System V IPC to do their work, and some may use the same keys. As these programs may be needed on several machines in the cluster, it was important to make them work with no interferences under DIPC. So it becomes necessary to allow two different kinds of IPC keys in a cluster: (1) *local keys*, which are used only in a single computer, and are usable only to processes on that computer, and (2) *distributed keys*, which are used to refer to the same IPC structure in the whole cluster.

A distributed key has a unique meaning in all the machines in the cluster, and referring to it should denote the same IPC structure. This means that when an IPC structure with a distributed key is created, there should be no other structure (local or distributed) with the same key in any other computer. To provide backward compatibility, different IPC structures with the same local key can exist in different computers. Local keys are the default key type: creating a distributed key requires the programmer to explicitly add an IPC_DIPC to other

usual flags while creating or gaining access to an IPC structure with a *xxxget()* system call. If it was not for the above requirement, DIPC would be used totally transparently. Even now the *only* thing the programmer has to do is using the IPC_DIPC flag.

5. dipcd Work Process

With DIPC, ordinary user programs interact only with the local kernel of the computer they run on. Requests of user programs for DIPC actions, whether synchronous ones like system calls that should be executed remotely, or asynchronous ones like trying to read from a shared memory with pages that are not available in the local computer, are routed inside the kernel. All such requests are put in a single linked list. The process *id* of the requesting user task is noted, and is used to find the original request when the results come back. This way the results can be delivered to the correct user program.

The kernel will in turn refer to the user space part of DIPC, *dipcd*, to actually fulfill these requests. This means that the presence of dipcd is transparent to user programs, and as far as they are concerned, the local kernel satisfies their requests.

dipcd is always waiting inside the kernel to collect these requests. Whenever it finds new ones, other parts of *dipcd* are activated to satisfy them. These parts get any necessary data (for example the parameters for a system call) to do the request from inside the kernel and return any results back to the kernel, where they will be delivered to the suitable user process. It is the *dipcd* that actually executes remote functions, transfers any data over the network, and decides which computer can read or write to a distributed shared memory. It also keeps necessary information about the IPC structures in the system and arbitrates between processes on different machines wanting to access the same structures at the same time.

It is apparent that there should be some provisions for *dipcd* to access the needed information in the kernel structures. A new system call, multiplexed with other System V IPC calls, is added to the Linux kernel to do just that. *dipcd* and other DIPC tools included in the DIPC package (like *dipcker*) use it to transfer data to and from the kernel. This new system call, used in DIPC programs as *dipc()*, is strictly for use by the *dipcd* and other related programs. Ordinary user programs should not use it.

Synchronising the creation of resources is important because it is possible that more than one process in the cluster want to create an IPC structure with the same key. These processes may be in the same computer or in different machines. So as to prevent unwanted interactions due to different processes trying to do the same thing, and possibly creating an inconsistent state in the whole system, the creation of an IPC structure should be done atomically, meaning that while one process is trying to create an IPC structure, no other process should try the same with an identical key.

The kernel parts of DIPC prevent more than one process in the same machine to attempt to create an IPC structure at the same time. They are serialised inside the kernel. Accomplishing the same effect across the cluster is made possible by a process responsible for this job: it is part of *dipcd* and will play the referee among requests from different machines and registers the necessary information about all the IPC structures in the system. It also controls the attempts to remove, or otherwise manipulate IPC structures. This process is called the *referee*.

There is a single referee in a cluster. All the machines in the same cluster should know on which computer the referee is currently running, and refer to it when needed. In fact, it is having the same referee that places two or more machines in the same cluster. In other words, a cluster is made of the machines that have the same referee. The referee address can be assigned by the system administrator, using the *dipc.conf* configuration file. Changing the address of a referee in a computer places that computer in another cluster. A machine that is running the referee process can also act as any other machine in the cluster.

6. Data and Information Transfer

In ordinary IPC, system calls are executed locally. Data provided by a process as parameters of a system call are copied inside the kernel and kept there. Each IPC call should return some result to the caller address space. The calling process will not be able to continue until it gets the results back. During this time it is waiting inside the kernel. The amount of time between making a call and getting an answer can vary greatly for different system calls and the IPC structure state. Some calls (e.g. *xxxctl()* with IPC_STAT command) return soon, Others (e.g a *semop()* call) can take very long or for ever to return. A local IPC system call thus requires two copies between the user and kernel address spaces, once from the user space to the kernel space, and once the other way around for the result.

RPC (Remote Procedure Call) is used to execute a system call on a remote computer. In order to ensure transparency, no process using DIPC should see any changes relative to the normal (local) IPC activities. Here too, data are copied from a process to

kernel's memory. *dipcd* then brings this data to its address space and transfers it over the network to the computer that is responsible for handling the request. This could be the computer on which an IPC structure was first created (the owner). The remote *dipcd* will copy the newly arrived data to the owner machine's kernel space. There are 3 copies and a network access for a process to simulate a system call in the destination computer. After this, the system call can be executed by *dipcd* in the remote kernel and the results will be sent back much the same way as described before: the remote kernel will copy the data to user space, so that *dipcd* can transfer them over the network. The *dipcd* at the original site of process receives this data and sends it to the local kernel. Now the data are copied from there to the original address space of the process.

The normal behaviour of programs willing to use System V IPC mechanisms for data exchange is like this: a process first creates an IPC structure by a suitable *xxxget()* system call using an agreed-upon key. Other processes can now use *xxxget()* to gain access to the same structure created before. In normal IPC, the first process causes the appropriate structures to be setup inside the kernel. Subsequent *xxxget()* calls merely return a numerical ID value, that can be used to refer to that structure. All these processes use and manipulate a single structure.

DIPC tries to mimic this situation as much as possible. Processes on different machines need to use the same key to be able to refer to the same IPC structure. In DIPC, when a process wants to create or gain access to an IPC structure with a certain key, the local kernel is first searched to see if that key is already used. This is quite like normal IPC. If the structure is found, the request is handled locally, with no reference to the referee. But if an IPC structure with that key is not found, then the referee should be consulted to find out if that key is already used in the cluster.

The referee searches its tables for the key, and tells the requester if the key was found or not, and if found, was it a distributed key or not. The referee can answer immediately if the key is present, or if the key is not present and no other machine has queried about it. After sending the information to the requesting computer, the referee expects a reply, informing it whether that machine locally created an IPC structure with that key or not, so that it can update its information, if necessary. But if the key was not found and the referee had already sent a message telling so to another machine, all further requests for that same key are not answered, until that other machine tells the referee if it created an IPC structure or not. When this information arrives, the referee can proceed to answer other waiting requests.

7. Distributed Shared Memory

DIPC provides strict consistency in its shared memory access. It means that a read will return the most recently written value. This is very familiar to programmers. There can be multiple readers of a shared memory (page) at a time, but only one writer at a time. The shared memory manager, part of *dipcd* running on the owner of the distributed shared memory, will receive the requests to read or write the whole segment or its individual pages. It will decide who will get the right to do the read or write, and if necessary, provides the requesting machine with the relevant shared memory contents.

MMU tables are changed to make the pages of a shared memory write-protected. The same is done to read-protect a page. Any process trying to read or write read-protected pages will encounter a page fault, and would sleep in the kernel. Before they can be readable or writable again, the new contents are brought over the network and replace the old ones. Now user processes can access them. DIPC can consider multiple virtual memory pages as one, managing and transferring all of them at the same time. This means that for any integer n, $n \geq 1$: <DIPC's page size> = $n \times$ <CPU's virtual memory page size>. A bigger DIPC page size will mean less overhead in transfers, and also makes it possible for computers with different native page sizes to be able to work with each other under DIPC. This value should be set to the maximum virtual memory page size in all computers of the cluster.

Two signals are used to inform processes of when they become readers or writers, so they can do any data conversions necessary in a heterogeneous environment. The signal currently used for reads is *SIGURG*, and *SIGPWR* is used for writes. They can be referenced in user programs with the names *DIPC_SIG_READER* and *DIPC_SIG_WRITER*.

All processes on the same machine have the same state regarding a shared memory. All of them can read or write it, or none of them can do any of the above. When a machine's access type to a shared memory changes, all processes on that machine are affected.

As mentioned before, DIPC can be configured to manage and transfer shared memories a page at a time. This is called *page transfer* mode. It allows different computers in the cluster to read or write different pages of a shared memory at the same time. DIPC could also consider the whole segment as an indivisible unit, in this case it is said to operate in segment transfer mode. Different nodes in a DIPC cluster, configured to use different transfer modes can work with each other, though they may not always get what they have asked for. The following

description shows how two computers with different transfer modes manage to work with each other. (1) The requesting computer and the shared memory manager both use pages; the requesting computer sends a request for a page, and will receive that page. (2) The requesting computer uses pages, while the shared memory manager uses segments; the requesting computer sends requests for one page from the manager, but the manager will send it the whole segment. The requester can access all the shared memory. (3) The requesting computer uses segments, while the shared memory manager uses pages; the requesting computer asks for the whole segment, but the manager only sends the page within which the access occurred. (4) The requesting computer and the shared memory manager both use segments; here the requesting computer asks for the whole segment, and gets it.

In case DIPC is configured in a segment transfer mode, then any transfer of shared memory contents would involve the entire segment. That is how DIPC can provide a *segment based* DSM. The reasons for allowing DIPC to be able to transfer whole segments are twofold. (1) It simplifies the working in a heterogeneous environment with different page sizes. (2) In some networks, the transfer time over the network is much less than the transfer setup time, so when the transfer is ready to begin, the amount that is transferred has very little significance.

The owner computer of a shared memory is its first writer, and it is always among the readers (if there are any readers). The way the owner computer is always present among the readers of a shared memory is like this. The owner always starts as the writer. When a request for read arrives, the owner is converted to a reader. If a process on another computer wants to write to the shared memory, then the owner will not have any access rights. As soon as a request for read arrives, the shared memory manager will place a request for read on behalf of the owner machine in front of the original read request. In this way, the owner will become a reader first, getting the shared memory contents from the current writer. It then provides the original reader (and possibly other requesters) with the contents.

In DIPC there is always one machine responsible for providing other computers with the shared memory contents. If there is a writer, this is the writer machine. If there are one or more readers, this is the owner machine. When a request to read or write fails remotely, the requesting computer will eventually find out about the problem through a time-out and send *SIGSEGV* (segmentation fault) to all processes that have attached the shared memory to their address space.

8. Experimental Results

We ran a number of experiments to measure the speed of distributed operations with DIPC. The test system consisted of two Pentium PCs connected together with a 10 Mb/s network. Table 1 shows the time taken to execute the *xxxget()* system calls (usually invoked only once in a program).

System Call	Time (Seconds)
semget()	0.04
msgget()	0.04
shmget()	0.04

Table 1. Creating the IPC structures.

Table 2 shows the resulting of remote execution of *xxxctl()* system calls [5], used for setting (IPC_SET) and retrieving (IPC_STAT) information about IPC structures.

System Call	Parameter	Calls/Sec
semctl()	IPC_STAT	49.50
semctl()	IPC_SET	50.00
msgctl()	IPC_STAT	49.00
msgctl()	IPC_SET	50.00
shmctl()	IPC_STAT	49.50
shmctl()	IPC_SET	45.00

Table 2. Controlling the IPC structures.

Table 3 shows the performance of the system when sending and receiving messages of different lengths. We measure how many times we could execute the system calls per second. The measured difference between sending and receiving messages were observed in many tests, and may be due to kernel-level implementation details.

System Call	Size (Bytes)	Calls/Sec
msgsnd()	1	97.14
msgrcv()	1	33.33
Msgsnd()	10	85.71
msgrcv()	10	29.41
Msgsnd()	100	97.14
Msgrcv()	100	33.33
Msgsnd()	1000	50.00
Msgrcv()	1000	50.00
Msgsnd()	4000	49.00
Msgrcv()	4000	49.00

Table 3. Sending and receiving messages.

Table 4 shows how many semaphore operations per second can be performed in our system.

System Call	Calls/Sec
semop()	48.00

Table 4. Semaphore operations.

Table 5 shows the amount of time it takes for two processes to access a DSM. Process 0 writes to the DSM, while process 1 reads from it. Both processes were running on the remote machine.

Process Number	DSM Size (Bytes)	Time (Sec)
0	1000	0.07
1	1000	0.06
0	7500	0.08
1	7500	0.07
0	25000	0.11
1	25000	0.10
0	50000	0.24
1	50000	0.24
0	80000	0.35
1	80000	0.36

Table 5. Writing from and reading to a DSM.

The set of programs written for the purposes of these experiments were considered worse-case distributed programs because they only exchange data, without any computation. In all tests, the increase in the amount of time needed to transfer data is proportional to the increase in the amount of data. Sending the data directly from user-space to user-space would be faster, because it would avoid some of the copying operations done by DIPC. This slowdown is the side-effect of the transparency of the system.

9. Concluding Remarks and Future Work

With DIPC, the task of writing a distributed application becomes similar to that of writing a local, parallel application. The programmer can develop an application that consists of many programs, running in parallel, to achieve a goal. The application should use IPC to share data and synchronise, and can be developed and tested on a single computer with a single CPU. DIPC then allows the same application to be run in a parallel and/or distributed environment with no changes. The drawback to such a scheme is that the data-transfer performance may be lower than in an application written specifically for a particular distributed hardware configuration.

DIPC is a heterogeneous distributed system that supports many CPU architectures. It retains backward compatibility with programs that use non-distributed IPC mechanisms. It can be ported to other CPUs supported by Linux, and also to other UNIX variants whose source code is publicly available.

Some possible areas for future work include porting DIPC to newer Linux kernels, porting DIPC to other UNIX variants, and enhancing its performance. Ensuring that DIPC and other distributed programming technologies, as included in the OSCAR or Rocks [6], for example, can work together is another worthwhile line of work.

The DIPC package, including the source codes, example programs, documentation and development tools, are available for download at http://www.cs.uwindsor.ca/~kamran/downloads.html.

References

[1] Claypool, M. Finkel, D., Transparent Process Migration for Distributed Applications in a Beowulf Cluster, *Proceedings of the International Network Conference* (*INC*), Plymouth, United Kingdom, July 2002.

[2] Gropp, W., Lusk, E., Sterling, T., *Beowulf Cluster Computing with Linux*, Second Edition, MIT Press, 2003.

[3] Karimi, K., Schmitz, M., and Sharifi, M., DIPC: A Heterogeneous Distributed Programming System, *The Third International Annual Computer Society of Iran Computer Conference (CSICC'97),* Tehran, Iran, December 1997. pp. 126-130.

[4] Sharifi, M. and Karimi, K., DIPC: The Linux Way of Distributed Programming, *Linux Journal*, Issue 57, January 1999. pp. 10-17.

[5] Silberschatz, A., Galvin, P.B., Gagne, G., *Operating System Concepts Seventh Edition*, John Wiley & Sons, 2004.

[6] Sloan, J.D., *High Performance Linux Clusters*, O'Reilley, 2005.

[7] Stevens, W.R., *UNIX Network Programming: Networking APIs, Sockets and XTI*, Second Edition, Prentice Hall, 1998

[8] Stevens, W.R. and Rago, S.A., Advanced Programming in the UNIX Environment, 2nd Edition, Addison Wesley Professional, 2005.

[9] Tanenbaum, A., *Distributed Operating Systems*, Prentice Hall, 1995.

[10] Linux Homepage: http://www.linux.org/